Efficient Fault Tolerance using Intel MPX and TSX

Efficient Fault Tolerance using Intel MPX and TSX

Oleksii Oleksenko*, Dmitrii Kuvaiskii*, Pramod Bhatotia*, Christof Fetzer*, Pascal Felber**
* Technische Universität Dresden, Germany
** University of Neuchâtel, Switzerland

***Abstract*—Hardware faults can cause data corruptions during computation, and they are especially harmful if these corruptions happen in data pointers. Existing solutions, however, incur high performance overheads, which is unacceptable for compute-intensive applications. In this work, we present an efficient fault-tolerance approach against hardware faults by exploiting the new extensions to the x86 architecture. In particular, we propose that Intel MPX can be effectively used to detect faults in data pointers, while Intel TSX can provide roll-back recovery against these corruptions. Our preliminary evaluation supports this hypothesis, and we estimate the average overhead to be roughly around 50%.**

I. Introduction

Critical components of the software stack such as operating systems, file systems, and databases are written in a low-level language (primarily, in C/C++) for improved performance and flexibility in memory management. However, the normal program flow can be arbitrarily corrupted by bit-flips in CPU or RAM during run-time. The resulting corruptions can lead to catastrophic consequences; especially, if the corruption happens in data pointers—as they can cause the loss of the whole data structure [1]. For instance, consider the case of Memcached—a widely used in-memory key-value store—if a fault happens in one of the hash table's pointers then an entire data bucket may be lost!

To mitigate data corruptions, a wealth of approaches based on duplicated execution [2] or thread-level redundancy [3] have been proposed. However, these existing solutions incur prohibitively high performance overheads, which makes them impractical to deploy for cyber-physical systems. This is particularly important for safety-critical applications that require both high assurance for fault-tolerance and minimal performance overheads for real-time constraints.

In the paper, we make a case for efficient fault tolerance against hardware faults. We aim to provide fault-tolerance for low-level systems code by leveraging the new ISA extensions in the x86 architecture. In particular, we aim to use the Intel Memory Protection Extensions (MPX) [4] to detect faults in data pointers, and use Intel Transactional Synchronization Extensions (TSX) [5] to provide recovery against faults.

In order to validate this early-stage idea, we measured the performance overheads of Intel MPX and Intel TSX separately on a set of applications from PARSEC benchmark [6]. The results demonstrate that the total performance overhead of the future implementation is expected to be 50% on average, which is a significant improvement over the state-of-the-art software-based solutions for fault-tolerance.

II. Assumptions

System model. We assume that the underlying hardware is based on the micro-architecture with support for memory protection and hardware transactional memory. In particular, our approach is based on the new generations of Intel architecture—starting Skylake—with respective support for memory protection via MPX [4] and transactional memory via TSX [5].

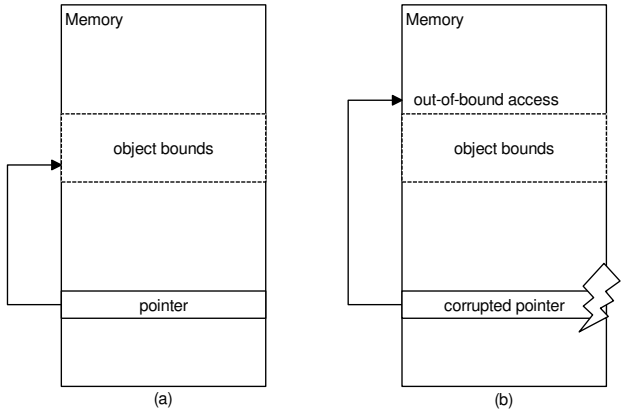

Fig. 1: The concept of fault detection using MPX: (a) correct state, (b) a fault in pointer causes object bounds violation.

Fault model. Since the recovery mechanism relies on re-execution, we assume that faults are transient in nature. Persistent faults can also be detected, but instead of recovery, they will cause a crash of the system after several unsuccessful retries (fail-stop semantics). Besides, our fault model does not impose any restriction on the number of bit-flips, and it is not restricted to Single-Event Upset (SEU) model [2]. In fact, higher bit-flip orders are likely to lead to the bigger changes in the pointer value, which in turn improves the fault detection probabilities. Moreover, we do not make any assumptions on memory protection, that is, we deal with errors in both DRAM and CPU.

III. Design

Our approach is based on two sets of extensions to the x86 architecture, namely on Intel MPX and TSX.

Fault detection. The fault detection principle is as follows: if a fault occurs in a pointer, the new value will violate the corresponding bounds with high probability (see Figure 1) and MPX can be exploited to detect such violations. MPX is an extension that adds a set of instructions for pointer bounds checking, as well as new registers for storing bounds data. MPX instruments all memory accesses (stores and loads) with bounds checks which are stored in the separate "bounds table". Even though its main goal is to protect from memory vulnerabilities in unsafe languages, such as buffer overflows, it can also be applied for hardware fault detection.

The exact probability of pointer fault detection depends on

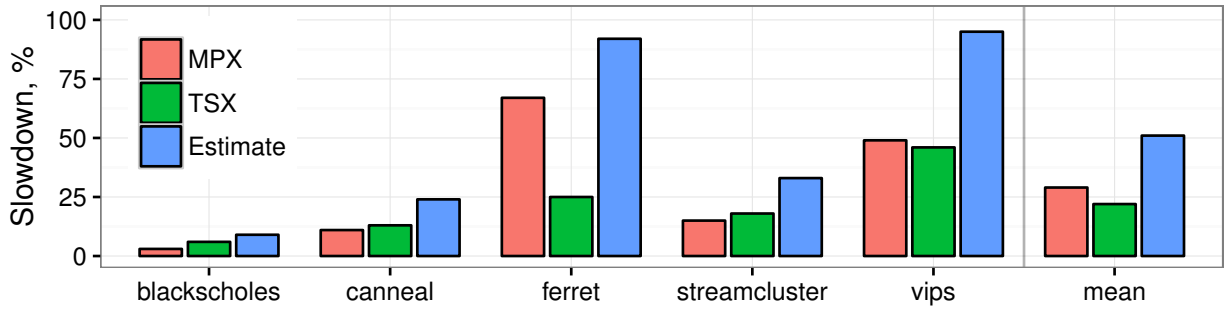

Fig. 2: Performance overhead over native execution for different x86 extensions

the object size and the bit flip order. For example, in case of 64-bit pointers (as in x86–64 architecture), probability of detecting a single bit flip is defined as follows:

$$P = 1 - \frac{\log_2 S}{64}$$

where P is the probability of detecting a fault and S is the size of the object in bytes.

Fault Recovery. The recovery mechanism is based on re-execution using TSX. Initially, TSX was designed as an alternative approach to the problem of thread synchronization. It, however, can be re-purposed for performing rollbacks, which gives us the ability to do fault recovery by re-executing a transaction (as, for example, done in HAFT [7]). Moreover, transactions in TSX are performed on hardware level using the L1 data cache, which makes them efficient (see IV).

Thus, TSX in combination with MPX for fault detection, gives us the ability to recover from hardware faults in pointers with reasonably low performance overheads. It works as follows: when MPX detects an error, it raises an exception and TSX reacts by performing a rollback of a currently executing transaction. This way, if a fault was transient and happened in the ongoing transaction, it will be recovered.

IV. Evaluation

In this section, we examine two key concerns of our approach:

- What is the expected total performance overhead?
- What are the overheads of the MPX (detection) and TSX (recovery) parts?

Experimental setup. We evaluated two considered CPU extensions with applications from the multithreaded benchmark suite PARSEC 3.0 [6]. All applications are built using Intel C++ Compiler 16.0.

For the performance evaluation, we used a 4-cores Intel Skylake processor operating at 2.7 GHz. It has 16 GB of RAM and is running Linux kernel 4.2.0. Each core has 32 KB L1 cache, 256 KB L2 cache, and 8192 KB of L3 cache are shared between all cores.

Results. Figure 2 presents the results of our preliminary performance measurements. The estimation here is done by summing up the overheads from MPX and TSX. The slowdowns are relative to the native versions with the same optimization flags. On average, the total overhead is expected to be 50%. The relative value, however, could be even lower, since the program may already be using MPX for security and fault detection will come for free.

In general, these results prove that combination of MPX and TSX used for fault tolerance may show better results than existing software-based solutions and, at the same time, will not require any additional or specialized hardware.

V. Conclusion

In this paper, we presented an approach on improving the performance of hardware fault detection and recovery in the restricted case of data pointer errors. Our hypothesis is that the efficient fault tolerance can be achieved by combining MPX and TSX extensions in the new x86 architecture, and our preliminary experiments show that it actually is the case—the average performance overhead is estimated to be 50%. However, our initial results should be interpreted with a grain of salt because: firstly, the fault-tolerance approach is restricted to data pointers only; and secondly, these overheads are based on micro-benchmarks. Nonetheless, we believe that these initial findings will foster discussion on a new direction for efficient fault-tolerance approaches based on the new ISA extensions.

Since our approach is restricted only to pointer faults, it must be combined with some other solutions to achieve complete fault tolerance. For example, in Elzar [8], pointer dereferences are one of the major bottlenecks, and removing them can significantly improve performance. Thus, our proposed approach and Elzar can be used together: the former for pointer dereferences and the latter for all non-pointer variables. We plan to explore this direction in our future work.

References

[1] Y. Aumann and M. A. Bender, "Fault tolerant data structures," in *FOCS*, 1996.

[2] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *CGO*, 2005.

[3] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *ISCA*, 2002.

[4] C. W. Otterstad, "A brief evaluation of Intel MPX," in *SysCon*, 2015.

[5] K. L. R. M. Yoo, C. J. Hughes and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *SC*, 2013.

[6] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS*, 2009.

[7] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT: Hardware-assisted fault tolerance," in *EuroSys*, 2016.

[8] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, "ELZAR: Triple Modular Redundancy using Intel AVX (Practical Experience Report)," in *DSN*, 2016.